VŠB TECHNICKÁ UNIVERZITA OSTRAVA
FAKULTA ELEKTROTECHNIKY A INFORMATIKY

# Absolvování individuální odborné praxe

Individual Professional Practice in the Company

Patrik Meixner

Bakalářská práce

Vedoucí práce: RNDr. Eliška Ochodková, Ph.D.

Ostrava, 2021

## Abstrakt

Tato práce pojednává o absolvování individuální odborné praxe ve společnosti ATLAS consulting s r.o, kde jsem pracoval jako člen frontendového týmu. Práce se zabývá od primárně vyvíjeného softwaru CODEXIS Green [1] a MDS Online [2], až po celkovou analýzu a rozebrání řešených úkolů a s němi spojenými technologiemi. Závěrem v této práci nalezneme hodnocení celkového působení na praxi, kde jsou shrnuty znalosti získané před praxí na akademické půdě či v rámci samostudia, tak i znalosti nabyté na praxi.

## Klíčová slova

React, JavaScript, TypeScript

## Abstract

This thesis describes the completion of individual professional practice in the company Atlas consulting s r.o., where I worked as a member of the frontend team. The thesis deals from the characteristics of the developed [3] and NMDS [**mds**] software, to the analysis of the solved tasks and related technologies. In conclusion, we will find an evaluation of the impact on practice, which summarizes the knowledge and overall skill development both acquired before the university study or even through self-development and the knowledge acquired in practice.

## Keywords

React, JavaScript, TypeScript

## Poděkování

Rád bych poděkoval všem, kteří mi s prací pomohli, všem kolegům a kolegyním ze společnosti ATLAS Consulting s.r.o. a vedoucímu mé odborné práce, paní RNDr. Elišce Ochodkové, Ph.D. jakožto vedoucí, za stravený čas u konzultací.

# Obsah

# Seznam použitých zkratek a symbolů

| | | |
|---|---|---|
| API | – | Application Programming Interface |
| TS | – | TypeScript |
| JS | – | JavaScript |
| JSX | – | JavaScript XMXL |
| CSS | – | Cascading Style Sheets |
| CSS-in-JS | – | Cascading Style Sheets in JavaScript |
| XML | – | Extensible Markup Language |
| HTML | – | Hyper Text Markup Language |
| DOM | – | Document Object Model |
| UX | – | User Experience |
| UI | – | User Interface |
| BE | – | Backend |
| FE | – | Frontend |

# Seznam obrázků

# Kapitola 1

# Úvod

V této bakalářské práci se budu zabývat krátkým seznámením se společností, využitými technologiemi, charakteristikou a přístupem k řešení zadaných úkolů a popisem průběhu mé odborné praxe.

Téma bakalářské práce, tedy individuální odborné praxe, jsem si zvolil na základě vlastní iniciativy a do jisté míry také zvědavosti ohledně tvorby internetových aplikací, které jsem se věnoval i ve svém osobním volnu.

Dle mého názoru je poněkud klíčové najít průnik mezi vědomostmi, získanými na akademické půdě, praktickými zkušenostmi a také začlenění jednotlivce do týmu, který pracuje na skutečném projektu. Tyto aspekty považuji za naprosto esenciální pro správný vstup do života programátora.

V následujících kapitolách krátce představím společnost ATLAS consulting s.r.o [3], ve které jsem mou odbornou praxi absolvoval, a software, který tato společnost vyvíjí a na jehož vývoji jsem se podílel. Ovšem primární částí této práce tvoří rozbor a řešení částí z jednotlivých zadaných úloh, se kterými jsem se za dobu mého působení ve společnosti setkal - od vytváření jednoduchých komponent, přes úpravy, opravy či refaktoring kódu až po implementaci nových funkcionalit v aplikacích MDS Online a CODEXIS Green.

V závěru celé bakalářské práce dojde ke shrnutí celé praxe a celkové zhodnocení praktických i teoretických znalostí, které jsem získal v průběhu studia a také znalosti, které jsem nabyl po absolvování této odborné praxe.

# Kapitola 2

# Seznámení se společností ATLAS consulting s.r.o.

ATLAS consulting spol. s r.o. člen skupiny ATLAS GROUP je ryze českou softwarovou společností, která se již od roku 1992 věnuje vývoji právních a manažerských informačních systémů a aplikací.

Společnost ATLAS software a.s. člen skupiny ATLAS GROUP byla založena v roce 2001 a od doby svého vzniku zajišťuje výhradní obchodní zastoupení společnosti ATLAS consulting spol. s r.o. Prodej, distribuci, školení a poradenství v oblasti software a hardware zajišťuje síť obchodních zástupců a školitelů. Péči o zákazníky a technickou podporu poskytuje společnost prostřednictvím vlastního klientského centra, zákaznické podpory a servisních pracovníků s celorepublikovou působností [4].

## 2.1 MDS Online

Aplikace Manažer datových schránek online [2] [stylizováno jako MANAŽER DATOVÝCH SCHRÁNEK] umožňuje správu datových zpráv v prostředí, na které jsou uživatelé zvyklí z e-mailových klientů.

Velkou výhodou aplikace je okamžitý přístup z běžného webového prohlížeče – kdykoli a odkudkoli. Ať už je uživatel v kanceláři, na cestách nebo pracuje formou home-office. Aplikace je vždy připravena k použití na počítači, mobilu ale i tabletu díky své responzivitě.

Navíc umožňuje neomezenou archivaci příchozí přijaté i odeslané pošty, administraci více datových schránek najednou, zaručuje snadné přihlášení bez nutnosti neustálého vyplňování údajů, nastavení úrovní oprávnění pro různé uživatele, umožňuje také elektronický podpis přímo v prohlížeči uživatele a mnoho dalšího.

## 2.2 CODEXIS

Codexis Green [1] [stylizováno jako CODEXIS Green] je bezesporu nejvýznamnějším produktem společnosti ATLAS consulting s.r.o. [3], který patří mezi nejrozšířenější právní informační systémy v České republice. Je zcela jedinečný díky svému zaměření, originálnímu zpracování po UX ale i UI stránce, pestré řadě funkcí a mnohým dalším užitečným funkcionalitám a vlastnostem. Čerpá veškerý svůj obsah z věrohodných zdrojů, kupříkladu Sbírky zákonů, Sbírky mezinárodních smluv a tím obsahuje zákony spadající do legislativy a judikatury České republiky a mimo jiné i legislativy a judikatury Evropské unie.

Tento software je určen především pro právníky a zástupce právnických profesí, finanční manažery, ekonomy, účetní, úředníky, manažery a specialisty, kteří k výkonu své práce potřebují snadný přístup k právním předpisům, zákonům a informacím k určitému tématu.

# Kapitola 3

# Pracovní náplň

## 3.1 Profil zaměstnance

Ve společnosti ATLAS consulting s.r.o. [3] jsem se podílel na vývoji prezentační vrstvy hlavních produktů, MDS Online a CODEXIS. Měl jsem možnost si vyzkoušet práci hned ve dvou týmech, kdy každý tým pracoval na jiném ze zmíněných produktů, podílet se na návrhu onoho řešení daných problémů, provádění revize zdrojových kódů ostatních kolegů a také testování výstupů. Má hlavní úloha na projektu CODEXIS byla rozšíření funkcionality, zjednodušení a revize zastaralého kódu, který se následně přepisoval do moderní podoby za použití těch nejmodernějších technologií, které v této době lze využívat.

Za použití knihovny React jsme dosáhli členění UI do znovupoužitelných modulů, které byly, pokud možno, co nejatomičtější. V rámci zachování jednotného stylu jsme vyvinuli další verzi již jednou vytvořené knihovny těchto atomických komponent a pomůcek, které jsme mohli dále využívat napříč projekty. Díky tomuto přístupu jsme dosáhli nejen jednotného stylu po vizuální stránce, ale i jednotného stylu co se týče logiky uvnitř komponent.

## 3.2 Organizace vývoje

Vývoj za pomocí scrum mastera probíhal agilně, každému ze spolupracovníků byly přiřazeny úkoly, které zpracoval a po jednotlivých sprintech, jenž trvaly 14 dní, probíhaly revize zdrojového kódu a následné případné konzultace.

### Plánování

Každý z týmů, kterých jsem byl součástí, byl rozdělen na dva celky. Jeden celek, jehož jsem byl součástí, tvořila menší skupina kolegů, kteří se zabývali vývojem uživatelského rozhraní a uživatelskou použitelností softwaru - tzv. frontend. Druhý celek tvořila taktéž menší skupina lidí, která prováděla

operace nad daty v programovacím jazyce Java - tzv. backend. Všichni členové týmu se participovali společně k dosažení zadaných úkolů v daném termínu, který byl každému z kolegů sdělen vedoucím týmu.

Práci na každém z úkolů jsem začínal podrobným rozborem a následným časovým odhadem, který jsem k řešení úlohy potřeboval. Následně jsem promyslel nejlepší možnou implementaci a rozdělení kusů kódu tak, aby vše bylo v souladu s konvencemi dodržované každým z frontendové části týmu. Před prvotní implementací jsem vše konzultoval s team leaderem či jiným z kolegů.

Každý z těchto úkolů jsem rozlišoval na dva samostatné celky - logickou a grafickou část. Logická část byla vyhrazená pro práci s daty, tj. transofrmací do požadovaného tvaru, filtrování a jinými operacemi. Grafická část se zabývala implementací uživatelského rozhraní za použití knihovny Styled components, ve které byly použity kaskádové styl a následnou integrací dat z logické části. Stylistická stránka byla z velké části konzultována s grafikem majícím na starosti grafické zpracování s responsivním designem pro stolní počítač, laptop, tablet a mobilní telefon, a následně předávána na kolegyni, která měla na starost právě práci s kaskádovými styly.

### Implementace

Již v úplném začátku praxe jsem byl představen verzovacímu systému Git, který je určen ke správě zdrojového kódu. Při vývoji v týmovém prostředí bylo nezbytné udržovat kontext s ostatními členy týmu, což bylo za pomocí tohoto nástroje jednoduché.

Celkovou implementaci jsem následně prokonzultoval ať už s kolegy či rovnou s vedoucím týmu pro případné nalezení nedostatků či chyb spojených s programátorskou nepozorností, čimž se dodržela kvalita softwaru.

### Testování

Po revizi bylo nutné nechat zdrojový kód projít testy založené na principu konec-konec (tzv. end to end testing) a zároveň také integračními testy. Následovalo testování dalším týmem, který nám dal zpětnou vazbu a oznámil nedostatky či schválil správnou funkčnost námi napsaného zdrojového kódu z pohledu běžného koncového uživatele.

### Nasazení

Po důkladném otestování a opravě nalezených chyb nasadil vedoucí týmu novou verzi aplikace s novou funkcionalitou, která se za jednotlivý sprint implementovala.

# Kapitola 4

# Použité klíčové technologie

V následující kapitole budou shrnuty esenciální technologie, které byly využity v průběhu vývoje.

## 4.1 TypeScript

TypeScript [5] je open-source programovací jazyk. Jedná se o nadstavbu nad jazykem JavaScript, která jej rozšiřuje o statické typování a další atributy, které známe z objektově orientovaného programování jako jsou třídy, moduly a další. Samotný kód psaný v jazyce TypeScript se kompiluje do jazyka JavaScript. Jelikož je tento jazyk nadstavbou nad JavaScriptem, je každý JavaScript kód automaticky validním TypeScript kódem.

## 4.2 JavaScript

JavaScript [6] dynamický, objektově orientovaný, především známý jako skriptovací jazyk primárně určený pro tvorbu webových stránek a aplikací. Podporuje jak objektové, tak i funkcionální a imperativní programovací paradigma.

## 4.3 React

React je open-source JavaScriptová knihovna, která se specializuje na tvorbu uživatelských rozhraní primárně vyvíjená a udržovaná společností Facebook. Tato knihovna využívá jisté principy a syntaxe, které jsou popsány v následujících bodech.

1. **JSX:** Jedná se o alternativní syntaxi, která je využívána k vytváření šablon používaných Reactem. Rozšiřuje principy ECMAScriptu o použití JSX struktur, umístěných v .js či .ts souborech. Jde o triviální *'syntactic sugar'* dělající kód čitelnější, tudíž není nutností jej používat [7].

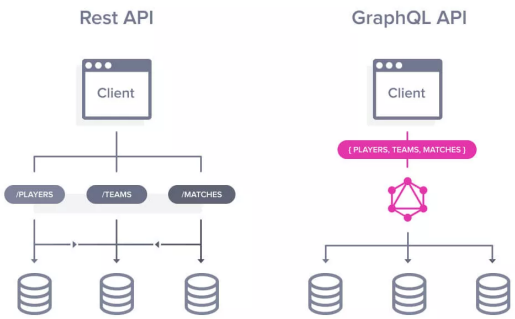
2. **Komponenty:** Komponenty [8], jakožto před-připravené, znovupoužitelné části kódu, umožňující rozdělit UI na nezávislé prvky o kterých je následně možno přemýšlet izolovaně. Pro lepší představu si lze komponenty vizualizovat jako lego částečky, které spolu vytvoří velkou stavebnici. Koncepčně jsou komponenty JavaScriptové funkce přijímající předem specifikované vstupy, nazývané „vlastnosti“ (anglicky properties), které jsou uvnitř komponenty statické a neměnné. Vracejí JSX objekty, které specifikují, co by se mělo na obrazovce objevit.

3. **Životní cyklus a stav:** React komponenty dodržují tzv. životní cyklus, jenž se skládá ze tří hlavních částí, kterými jsou zavedení do DOMu, aktualizování (tzv. přerenderování) a případně odebrání z DOMu. Stav [9] komponenty je JavaScriptový objekt obsahující informace, které jsou v komponentě dále využívány a jsou variabilní v závislosti na životním cyklu komponenty.

4. **React Hooks:** Háčky (anglicky Hooks) [10] jsou funkce, které umožňují obohatit komponentu o určitou funkcionalitu. Dokáží reagovat na stav a životní cyklus komponent. Hooky fungují pouze ve funkcích, nikoli ve třídách čímž lze dosáhnout programování dle funkcionálního programovacího paradigma. Díky hookům je možnost sdílení logiky, které se řeší návrhovými vzory High order Component a mimo jiné také Function as Child (tzv. Render Props) [11].

## 4.4 GraphQL

GraphQL [12] dotazovací jazyk pro API moderních aplikací. Klientem specifikované textové řetězce s dotazem jsou interpretované serverem, což vrací data v definovaném formátu. GraphQL přináší kompletní a srozumitelný popis dat, díky čehož klient pomocí tzv. queries získává možnost vyžádat přesně ta data, která potřebuje pro následné vykonávání dalších operací. Zároveň má klient možnost tato data měnit pomocí takzvaných mutací. V porovnání například s Rest API, které využívá z pravidla více koncových bodů (tzv. endpointů), GraphQL na druhou stranu využíva endpoint pouze jeden (viz obrázek č.5.2).




Obrázek 4.1: GraphQL vs Rest [13]

## 4.5 Apollo Client

Apollo Client [14] je platforma, postavená na základním open source klientovi GraphQL. Poskytuje kompletní správu stavu v JavaScriptových aplikacích. S jeho pomocí stačí specifikovat GraphQL dotaz a o vše ostatní se postará právě Apollo Client - tj. dotázání backend služeb. Po zmíněném dotázání backend služeb se vrátí vývojářské funkce a data, které jsou následně uloženy do mezipaměti (tzv. cache) a ve stejnou chvíli jsou aktualizované i komponenty využívající Apollo klienta.

## 4.6 Styled Components

Styled Components [15] je knihovna využívající principy a implementace CSS stylů uvnitř samotného JavaScriptového kódu (tzv. CSS-in-JS), díky čemuž dává uživateli možnost komponenty úplně zapouzdřit a také dědit CSS styly z jiných komponent.

## 4.7 Codegen

Codegen [16], CLI nástroj, který automaticky generuje části zdrojového kódu, které je možnost dále využívat. Nabízí odstranění nutnosti explicitní definice typů a metod spojených s backend službami, které jsou implicitně vygenerovány po analýze a následném zparsování předem definovaného GraphQL schématu.

## 4.8 React Spring

React Spring [17] je knihovna, která se specializuje na fyzické animace, reprezentující moderní přístup k tvorbě animací. Tato knihovna poskytuje různé funkce, které pokrývají naprostou většinu animačních potřeb souvisejících s animováním UI.

## 4.9 Codegen

Codegen [16], CLI nástroj, který automaticky generuje části zdrojového kódu, které je možnost dále využívat. Nabízí odstranění nutnosti explicitní definice typů a metod spojených s backend službami, které jsou implicitně vygenerovány po analýze a následném zparsování předem definovaného GraphQL schématu.

# Kapitola 5

# Řešení zadaných úkolů

V první fázi jsem měl za úkol pracovat převážně na nové funkcionalitě softwaru CODEXIS [1], kde jsem společně s jedním členem backendového týmu implementoval uživatelské prostředí pro vzájemnou spolupráci více uživatelé v aplikaci. Ve fázi druhé jsem byl přesunut do druhého týmu, který měl na starost software MDS Online, kde jsem taktéž vyvíjel novou funkcionalitu, a sice štítkování zpráv, pravidla pro nově přijaté zprávy, dále práci s aktualizací Apollo cache, animování modálních oken a jiné. V následujících kapitolách přiblížím časovou náročnost ,specifikaci, analýzu a řešení těchto úkolů.

## 5.1 Časová náročnost jednolivých úkolů

### CODEXIS

- Workspace [3 týdny]

### MDS Online

- Štítkování zpráv [6 dní]
- Animace [4 dny]
- Pravidla pro nově přijaté zprávy [10 dní]
- Aktualizace Apollo cache [3 dny]
- Tištění zpráv a příloh [3 dny]

## 5.2 Workspace

### Specifikace a požadavky

Cílem této úlohy bylo rozšíření a menší přepracování již zaběhlého doplňku Workspace, který slouží ke sdílení pracovní plochy, dokumentů či k plánování událostí prostřednictvím kalendáře aj. Jednalo se o jeden z nejpokrokovějších doplňků, které aplikace Codexis za poslední léta dostala.

Mým primárním úkolem bylo přepracování hlavní stránky (tzv. dashboardu) doplňku Workspace, který obnášel zobrazení sdílené historie, oblíbených dokumentů, kalendáře a jeho událostí a také nástěnku, na které uživatelé můžou přidávat příspěvky, sdílet své názory s ostatními uživateli a reagovat na názory jiných.

### Časová náročnost

Vývoj přepracování tohoto doplňku zabral valnou většinu mé docházky v zimním semestru, přibližně 3 týdny čistého času, kdy uzávěrka byla na rozmezí nového roku. Tento čas byl ovlivněn spoluprácí s backendovou částí úlohy, na které pracoval můj kolega.

### Analýza a návrh

Velkým problémem sdílení dat v reálném čase mezi více uživateli bývá synchronizace. Proto bylo zvoleno GraphQL a Apollo Client, který díky svým funkcím a možnostem umožnil jednoduchou aktualizaci dat uživatelů, kterým měl být jejich obsah aktualizován. Obrovskou výhodou tohoto přístupu byla poměrně jednoduchá práce s již zmíněnou aktualizací a také práce s mezipamětí.

Jelikož se jedná o rozšíření a úpravu funkcionality, která již byla v minulosti implementována, jsem měl možnost se inspirovat starším kódem, ale zároveň se ponaučit z chyb, které byly v minulosti implementovány a taktéž využít novější technologické postupy.

Struktura logické i prenzentační vrstvy jsem navrhnul tak, aby byla rozdělena do spousty funkcionálních komponent, kdy k valné většině z nich měla být přidána další funkce nesoucí logickou část, kterou dále předávala do dalších komponent - tento přístup je znám jako High Order Component spolu s Render Props.

Návrh složitějších komponent a částí hlavní strany doplňku Workspace jsem konzultoval s vedoucím frontendového týmu.

### Implementace

Na vyřešení této úlohy jsem postupoval tak, jak jsem navrhnul spolu s vedoucím frontendového týmu.

První jsem rozčlenil jednotlivé větší celky, podle kterých jsem rozvrhnul základní strukturu hlavní strany doplňku Workspace.

**Jednalo se celkem o pět sekcí:**

- Hlavička, která poskytovala informace o právě zvoleném pracovním prostoru spolu s kontrolními prvky
- Sekce mající za úkol přesměrovávání uživatele na jinou část aplikace
- Panely nesoucí informace ohledně dokumentů a kalendář
- Levý panel, jenž byl rozšířen o archivované pracovní prostory
- Nástěnka, na které byla možnost sdílet obsah s ostatními uživateli stejného pracovního prostoru

Část těchto sekcí byla umístěna v jedné komponentě, pojmenované Dashboard, sloužící pouze jako kontejner pro jednodušší implementaci kaskádových stylů. Zbylé sekce byly umístěny v nadřazené komponentě, která využívala již zmíněný Dashboard.

V následující části budou v krátkosti dvě sekce interpretovány a alespoň nastíněn způsob jejich implementace.

### 5.2.1 Hlavička

Komponenta hlavička jsem se rozhodl z důvodu responzivity rozdělit na mobilní a desktopovou část, kdy v závislosti na zařízení, na kterém měla být vykreslena, se vykreslila právě daná komponenta. Toto jsem docílil za použití interní komponenty Device, jenž rozpoznává zařízení na základě výšky a šířky displeje a má možnost vykreslovat komponenty v závislosti, zdali zařízení, které má definované, spadá do daného rozlišení.

```
export const Heading: FC<Readonly<HeadingProps>> = (): Readonly<ReactElement> => (
  <>
    <Device devices={['tabletHorizontal', 'laptop', 'desktop']}>
      <HeadingView.Desktop />
    </Device>
    <Device devices={['mobile', 'tabletVertical']}>
      <HeadingView.Mobile />
    </Device>
  </>
);
```

Listing 5.1: Rozdělení hlavičky podle vykreslovacího zařízení

Následně jsem implementoval kontrolní tlačítka, jenž slouží pro akce jako je kopírování daného pracovního prostoru, opuštění, smazání, editace a archivace. Pro příklad popíši funkci pro smazání pracovního prostoru (viz výpis č. 5.2).

Vytvořil jsem si novou funkci onDelete, v níž jsem si definoval dva podbody. Zobrazení modálového okna pomocí funkce showModal a konkrétní odstraňovací funkci onDeleteModal, jenž byla využita uvnitř zmíněného modálového okna při potvrzení.

Z důvodu, že pracovní prostor může smazat pouze vlastník, bylo nutné ošetřit tuto funkcionalitu - pro tento případ jsem porovnával proměnnou userProfile.login, získanou z GraphQL, jenž mi dávala informaci ohledně právě přihlášeného uživatele. Tuto proměnnou jsem poté porovnal s vlastníkem pracovního prostoru, jenž byl získán taktéž z GraphQL dotazu.

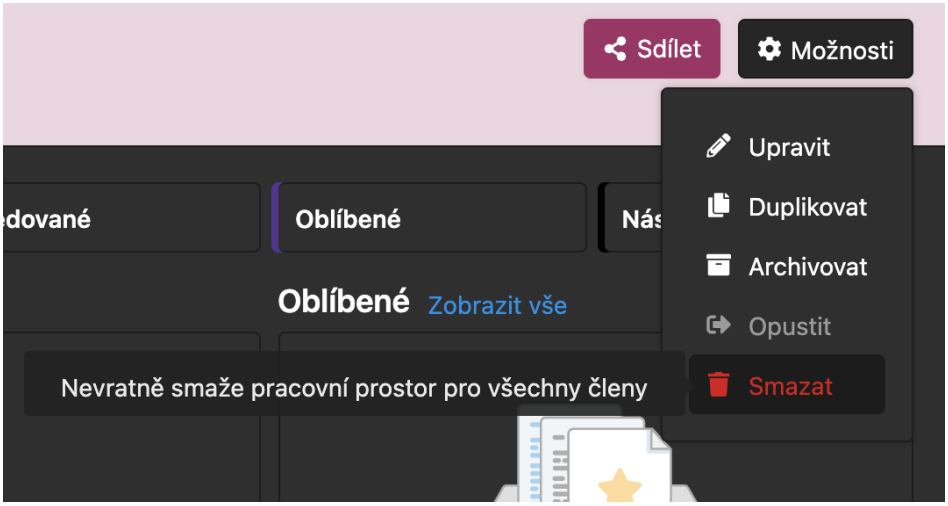
```
const { workspace, workspaceId, setWorkspaceId, refetch } = useWorkspace();
const { userProfile } = useGlobal();
const { showModal } = useModal();
const [deleteWorkspace] = useDeleteWorkspaceMutation();
const { sendMessage } = useMessage();
const isOwner = userProfile.login === workspace?.owner;

const onDelete = () => {
      const onDeleteModal = async () => {
         if (workspaceId && isOwner) {
            const response = await deleteWorkspace({ variables: { workspaceId } });

            if (response.data) {
               sendMessage('success', 'Pracovní prostor byl odstraněn');
               setWorkspaceId(null);
               refetch({ workspaceId });
            }
         }
      };
      showModal(
         `confirmationModal`,
         <ConfirmationModal
            confirmButtonProps={{ onClick: onDeleteModal }}
            text={`Opravdu si přejete smazat pracovní prostor? Tato akce je nevratn
                á!`}
            title={`Smazání pracovního prostoru`}
         />
      );
};
```

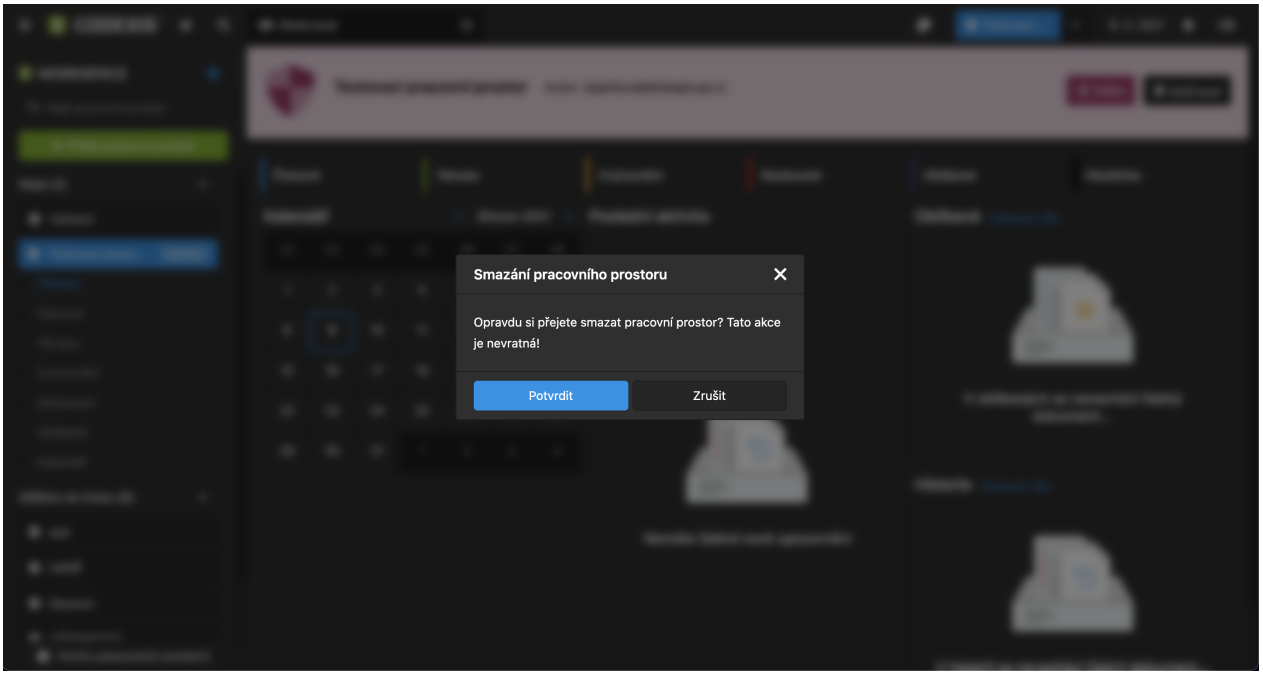
Listing 5.2: Funkce pro smazání pracovního prostoru

Před vyvoláním onoho modálního okna bylo potřeba z nabídky kontrolních tlačítek vybrat tlačítko pro smazání pracovního prostoru, jenž upozorňuje, že tato akce je nevratná.




Obrázek 5.1: Nabídka kontrolních tlačítek

Po kliknutí na tlačítko pro smazání pracovního prostoru bylo vyvoláno modální okno, jenž pro jistotu, že uživatel opravdu chtěl smazat pracovní prostor, znovu připomene, že se jedná o nevratnou akci a vyzve uživatele k potvrzení či zrušení akce.




Obrázek 5.2: Modální okno pro smazání

Po potvrzení akce je pracovní prostor smazán, všichni uživatelé jsou z pracovního prostoru vyloučeni a notifikováni o této události.

### 5.2.2 Sekce pro přesměrování

Pro sekci pro přesměrování byl využit návrhový vzor Higher-Order Component. Byl vytvořen Hook [10] useRedirectSection, jenž nesl celou logiku této sekce. Následně byla vytvořena komponenta RedirectSectionView nesoucí pouze vizuální část komponent. Tyto dvě separátní části byly spojeny pomocí interní funkce wrap, jenž rozbaluje funkcionalitu do vizuální komponenty, a vytvořily komponentu *RedirectSection* (ukázka použití viz. výpis č. 5.3), jenž byla již hotová komponenta připravená na použití kdekoliv v aplikaci.

```
export const RedirectSection = wrap(RedirectSectionView, useRedirectSection);

export const useRedirectSection = () => {
    const { counts } = useWorkspaceCounts({ ...followed });
    ...onTopicsTileClick,

    return { counts, onTopicsTileClick }
}

export const RedirectSectionView: FC<RedirectSectionProps> = (props) =>
     <StyledRedirectSection>
        ....
        <StyledRedirectSectionTile>
           <Tile
              dataId={‘topics‘}
              title={‘Témata‘}
              color={theme.color.success.alpha}
              count={counts.get(EIconCounters.FILES)!}
              onClick={onTopicsTileClick}
           />
        </StyledRedirectSectionTile>
        ....
     </StyledRedirectSection>
```

Listing 5.3: Higher-Order Component sekce pro přesměrování

## 5.3 Štítkování zpráv

### Specifikace a požadavky

Po přesunutí na nový projekt mi byl zadán úkol zpracovat frotendovou část funkcionality, jenž řešila štítkování zpráv, tedy označování zpráv uživatelem specifikovanými barvami s popisky pro přehlednější a jednodušší třízení zpráv.

Spolu se samotným označováním zpráv bylo mým úkolem vytvořit rozhraní pro vytváření upravování a mazání těchto štítků.

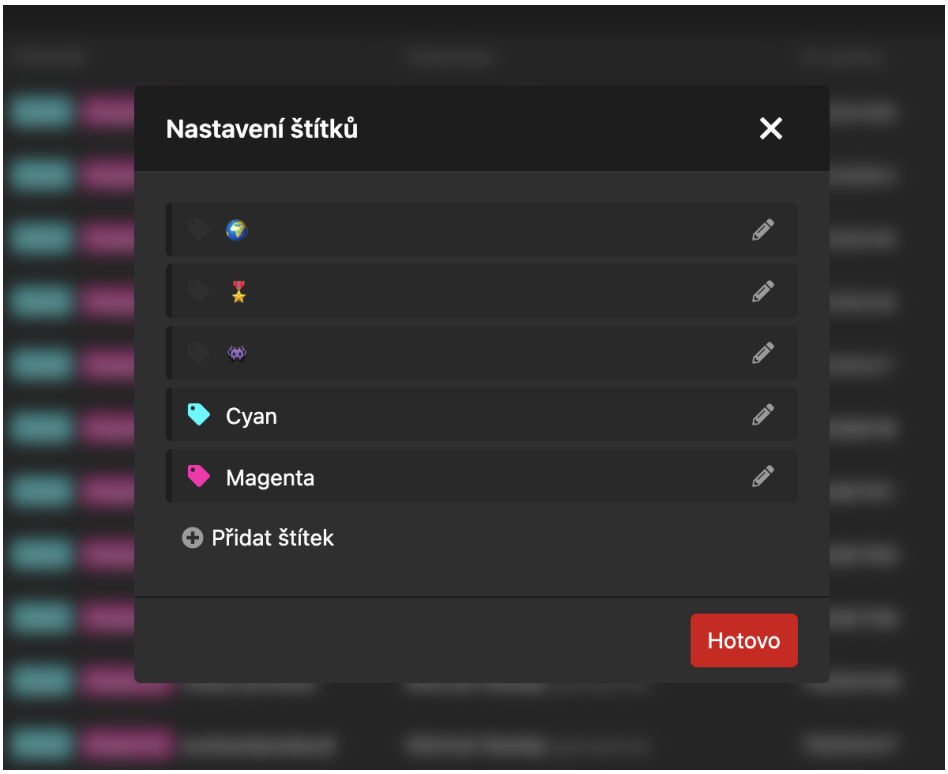
### Časová náročnost

Vývoj této části zabral bezmála šest dní přičemž první tři dny probíhal vývoj, další jeden den jsem kód refaktoroval do přijatelnější podoby a následné dva dny probíhalo testování a opravování nedostatků a chyb.

### Implementace výpisu a vytvoření štítku

Nejprve jsem implementoval modálové okno, jenž poskytovalo výpis všech štítků a také v něm bylo možné štítek vytvořit. Při postupu jsem využil techniku, kterou jsem se naučil v předchozím úkolu, tedy využití hooku a zobrazení jednotlivé komponenty.

Hook poskytuje data a metody potřebné pro danou funkcionalitu a zobrazení obsahuje Styled Components [15], které zajišťují vzhled modálového okna a jeho obsahu.




Obrázek 5.3: Výsledné modálové okno s výpisem štítků

### Implementace editace a smazání štítku

Jako následující bod jsem implementoval editaci a smazání štítku vytvořením dalšího modálového okna, které bylo strukturálně naprosto totožné jako modálové okno předchozí, ovšem s jinými funkcemi a jiným vzhledem Styled Components [15] (viz. obrázek č. 5.5).

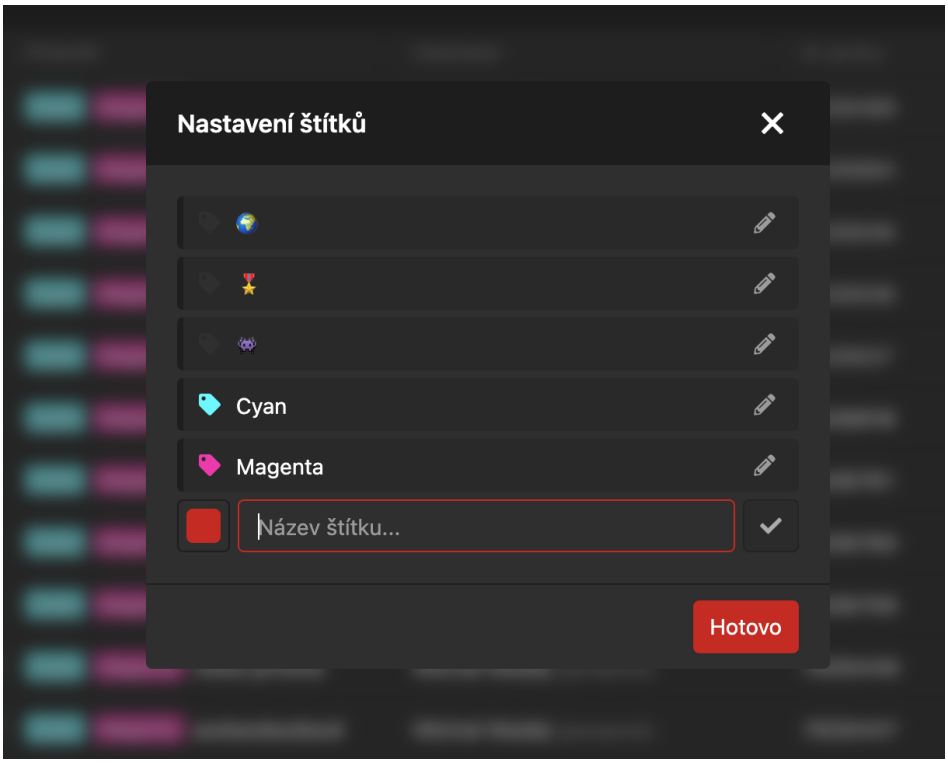
### Implementace označování zpráv

Označovování zpráv byla poněkud zapeklitější funkcionalita. Nejprve jsem, stejně jako v předešlých případech, vytvořil hook a zobrazení komponenty.

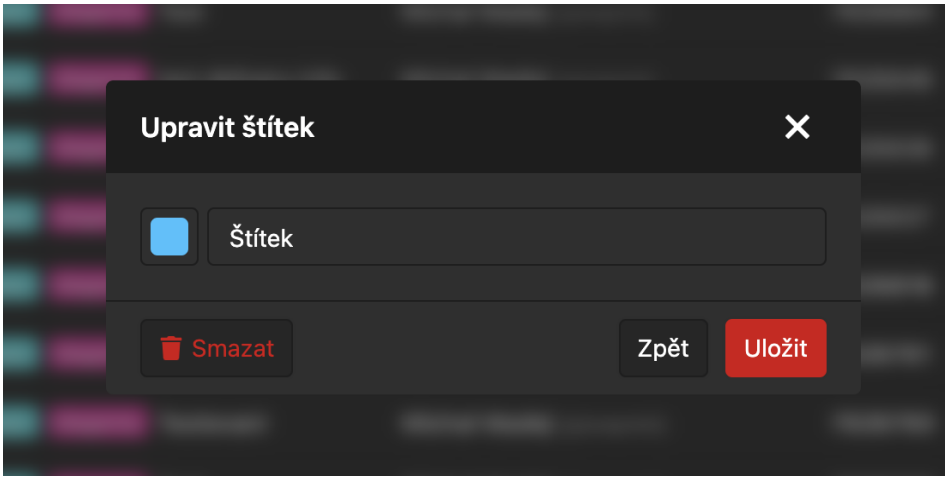
Označování probíhalo prostřednictvím kontextové nabídky ukryté pod tlačítkem. Ke každému štítku byla přiřazena barva, text a zaškrtávací pole (tzv. checkbox), který nabýval třech stavů - úplně zaškrtlý, částečně zaškrtlý a nezaškrtlý.

V hooku a jeho funkcionalitě jsem musel myslet na to, že je třeba zjišťovat informace ohledně toho, zdali nějaký z označených zpráv je označená nějakým štítkem pro zobrazení určitého stavu v checkboxu. Toto jsem zjišťoval po najetí na tlačítko označování (viz obrázek č. 5.6)

V případě, že všechny zaškrtlé zprávy jsou označeny stejným štítkem, nabývá checkbox stavu úplně zašktlý. Dále v případě, že alespoň některá ze zaškrtlých zpráv je označená štítkem, pak



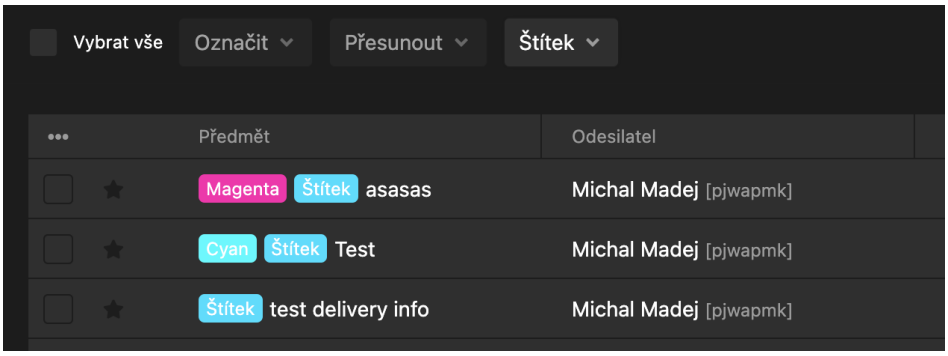

Obrázek 5.4: Výsledné modálové okno s kontrolními prvky pro přidání štítku



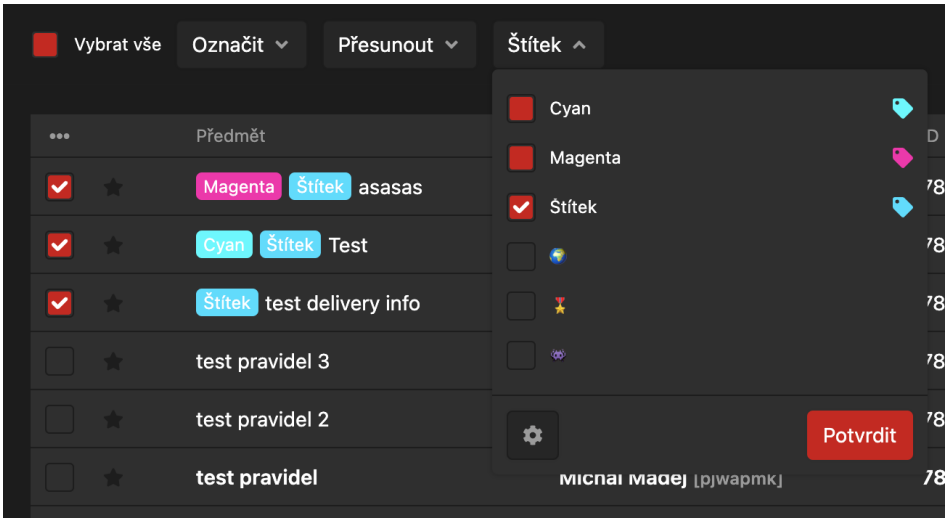

Obrázek 5.5: Výsledné modálové okno s kontrolními prvky pro editaci a smazání štítku

checkbox nabývá stavu částečně zaškrtlé. A v neposlední řadě, v případě, že ani jedna zpráva není označena štítkem, checkbox štítku nabývá hodnoty nezašktlý (viz obrázek č. 5.7).

Po kliknutí na checkbox příslušící k danému štítku se zaškrtlé zprávy tímto štítkem označí a checkbox změní stav.




Obrázek 5.6: Tlačítko pro vyvolání kontextové nabídky označování zpráv




Obrázek 5.7: Kontextová nabídka s označenými zprávami

V levém spodním rohu kontextové nabídky je také vidět tlačítko nastavení, které po kliknutí vyvolá modálové okno s výpisem štítků a možností přidání štítku.

## 5.4 Animace

### Specifikace a požadavky

Za úkol bylo vytvořit rychlé, plynulé a snadno použitelné animace pro modálové okna za použití knihovny React Spring. Jednalo se o různé animace, hlavně pro vytváření a odebírání modálových oken z DOMu.

### Analýza a návrh

Modálové okna již měly vlastní implementaci, ovšem byly provizorně animovány za použití keyframes, které poskytuje knihovna Styled Components. Bylo zapotřebí poupravit a do jisté míry přepracovat tuto implementaci a celkové chování animací, přidat nové funkce pro určování animací a také využít knihovnu React Spring.

Animace jsem navrhl tak, aby v každé fázi měly volitelné vlastnosti, tedy aby například v případě modálového, jenž se má vysouvat a zasouvat z pravé strany okna mohlo dojít k tomu, že při vykreslování se objeví z pravé strany (tj. směrem doleva) a následně při odebírání z domu se zpátky do pravé strany (tj. směrem doprava). Zároveň bylo potřeba ošetřit rychlost a svižnost.

### Implementace

Jelikož se jednalo o animování modálových oken, které se zpravidla skládají z jakéhosi tmavého pozadí, na kterém je vykreslen právě obsah daného okna, rozvrstvil jsem si tyto animace na dva hlavní sektory - pozadí (tzv. *overlay*) a obsah (tzv. *children*).

Každý z těchto sektorů měl stejné vlastnosti - směr animace, nastavení animace (rozostření, průhlednost, aj.) a konfigurace animace (jak agresivní má animace být). Nastavení a konfigurace animací mají navíc možnost určit tyto vlastnosti pro vykreslování a odebírání z DOMu.

Pro každý ze sektorů jsem využil funkci *useTransition*, kterou poskytovala knihovna **React Spring**. Tato funkce přijímala již zmíněné vlastnosti a také přepínač *isModalOpen*, který měl za úkol zjišťovat, zdali je modálové okno otevřené či nikoliv - podle tohoto přepínače se určovalo zdali je modálové okno právě přidáváno či odebíráno do DOMu.

Tyto jednotlivé sektory na sebe byly navázány pomocí funkce *useChain* (použití viz. výpis č. 5.4), kterou taktéž poskytovala knihovna **React Spring**. Tato funkce přijímala přepínač *isModalOpen*, pole referencí na každý ze sektorů a také pole časů a zajišťovala návaznost jednotlivých sektorů na sebe sama. Ve zkratce, při přidávání modálového okna do DOMu se pozadí modálového okna vykreslilo první a v návaznosti na to se vykreslil obsah modálového okna. Na druhou stranu, tedy při odebírání modálového okna z DOMu, se první odebral obsah a v návaznosti na to se odebralo pozadí, což zajistilo příjemný a čistý dojem.

```
useChain(
      isModalOpen ? [overlayRef, childrenRef] : [childrenRef, overlayRef],
      [0, isModalOpen ? 0.1 : 0.225]
   );
```

Listing 5.4: Použití funkce useChain poskytnuté knihovnou React Spring

## 5.5 Pravidla pro nově přijaté zprávy

### Specifikace a požadavky

Začátkem dalšího dvoutýdenního sprintu jsem dostal za úkol vytvořit UI pro pravidla pro nově přijaté zprávy - tedy co se má stát se zprávami, které jsou uživatelem nově přijaty. Jednalo se o UI pro zobrazení přehledu pravidel, vytvoření, editaci, kopírování, smazání a spuštění daného pravidla.

Jednotlivé pravidlo mělo nést název, pro kterou datovou schránku je použito, dále podmínky, pod kterými má být spuštěno (např. odesílatel obsahuje "*jméno-odesílatele*") a v neposlední řadě akce, které se mají provést (např. přidat štítek).

### Analýza a návrh

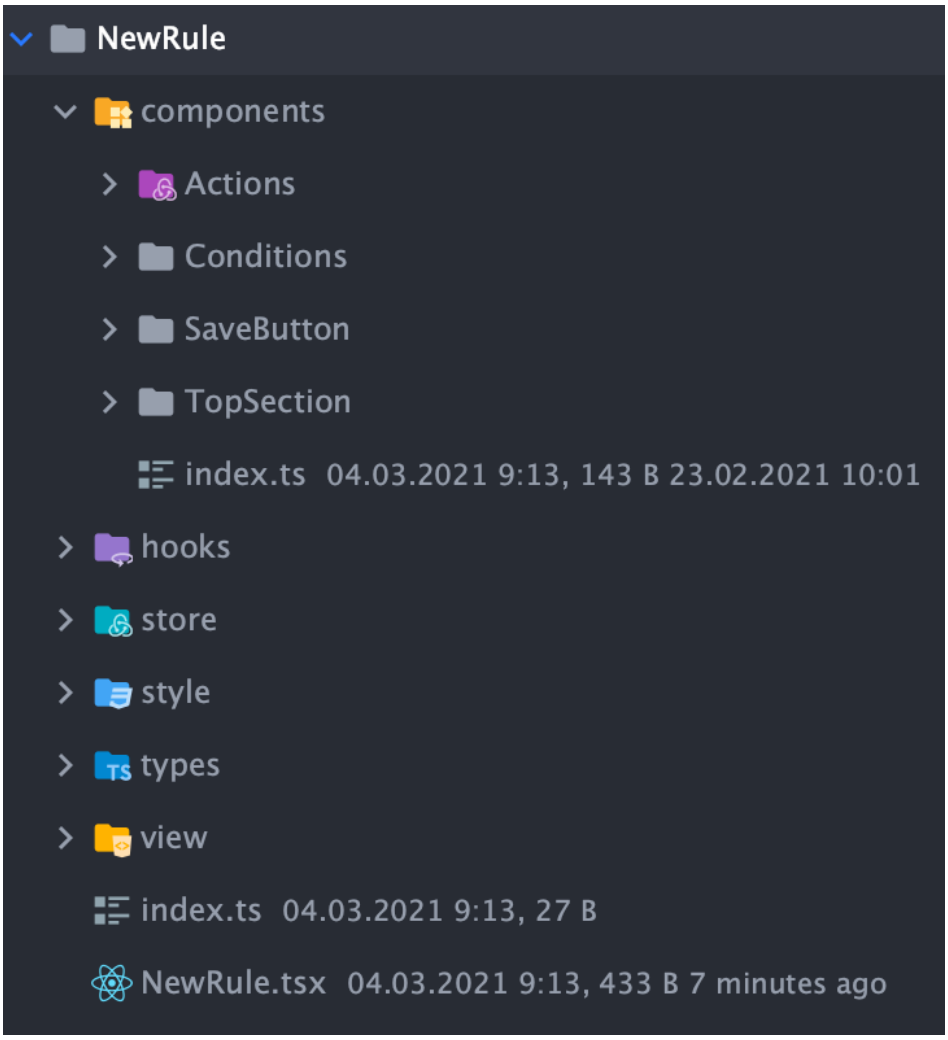
Úloha se skládala ze dvou podúloh - modal výpisu pravidel a modal pro práci s jedním pravidlem.

Bylo mi jasné, že výpis pravidel bude poněkud jednodušší, jelikož se jednalo pouze o výpis pole komponent. Již v analýze jsem pochopil, že druhá podúloha bude poněkud složitější, jelikož bylo zapotřebí vytvořit modálové okno, jenž bude zaopatřovávat všechny funkce jako editace, kopie a vytvoření nového pravidla, aby nedocházelo k duplicitě co se týče kódu a aby vše bylo jednoduše testovatelné.

V první řadě jsem si stanovil, že každá z výše uvedených částí bude obsažena ve vlastní komponentě. Rozdělil jsem si vše do čtyř sekcí - vrchní část, podmínky, akce a tlačítko na uložení pravidla (viz. obrázek č. 5.8).

Zároveň bylo nutné uvažovat nad tím, že pravidla i akce mohou postupně vykreslovat odlišné komponenty v závislosti na předem zvolených parametech - tj. uživatel zvolí podmínku "*osobní doručení*", které má vykreslovat Checkbox a nebo uživatel zvolí podmínku "*odesílatel*", které má vykreslovat Select a Input. Proto bylo zapotřebí vytvořit mezikomponentu, která tuto událost brala na vědomí.

Všechny informace ohledně jednoho pravidla jsem se rozhodl uchovávat ve Formiku.




Obrázek 5.8: Struktura souborů druhé podúlohy

### Implementace

V následující sekci stručně popíši implementaci složitějšího podúkolu, tedy toho druhého. Jako nejlepší část tohoto podúkolu pro demonstraci jsem si zvolil **Akce**.

Jako první věc, kterou jsem si připravil, byla komponenta **NewRuleActionsView** [viz. výpis č. 5.5], která obalovala název sekce, komponentu listu akcí, ke které se za chvíli dostaneme a také tlačítko pro přidání další akce. K této komponentě, stejně jako k ostatním, náležel hook nesoucí funkce a hodnoty využité v komponentě.

```
const NewRuleActionsView = ({ onAddAction, canAddAction }) => (
    <StyledNewRuleModalAction>
        <Text element="strong">
            {translate(translation.settings.action)}
        </Text>
        <NewRuleActionList />
        {canAddAction && (
            <SimpleButton onClick={onAddAction}>
                <CustomIcon name="plusCircle" />
                <Text>{translate(translation.settings.addAction)}</Text>
            </SimpleButton>
        )}
    </StyledNewRuleModalAction>
);
```

Listing 5.5: Komponenta akcí

Následně jsem si vytvořil další, již zmíněnou komponentu, která obstarávala vykreslení všech akcí - tzv. **NewRuleActionList** [viz. výpis č. 5.6]. Proměnná *actions* nesla informace o všech akcích, které mají být vykresleny. Proto jsem touto proměnnou mapoval a předával informace do nadcházející komponenty **NewRuleActionItem** vykreslující právě jednu z akcí.

```
const NewRuleActionListView = ({ actions, actionOptions }) => (
    <StyledNewRuleActionList>
        {actions.map((action, index) => (
            <NewRuleActionItem
                index={index}
                action={action}
                actionOptions={actionOptions}
            />
        ))}
    </StyledNewRuleActionList>
);
```

Listing 5.6: Komponenta listu akcí

V této fázi jsem začal řešit již zmiňované kondicionální vykreslování komponent na základě předešle zvolených hodnot. Toto vykreslování probíhá uvnitř komponenty **NewRuleActionValueComponentView** (viz. výpis č. 5.7), kde se na základě jednoduchých funkcí zjišťuje jaká z komponent má právě být vykreslena. Každá z těchto komponent má vlastní zobrazení (tzv. view) a vlastní funkcionalitu obsaženou v hooku. Zároveň každá komponenta přijímá stejné properties, které jsou jí předávány.

```
export const NewRuleActionValueComponentView = ({ ...props }) => {
    if (shouldActionRenderFolderSelect(props.actionType!)) {
        return <NewRuleActionFolderSelect {...props} />;
    }

    if (shouldActionRenderInput(props.actionType!)) {
        return <NewRuleActionValueInput {...props} />;
    }

    if (shouldActionRenderTagSelect(props.actionType!)) {
        return <NewRuleActionTagSelect {...props} />;
    }

    return <></>;
};
```

Listing 5.7: Komponenta listu akcí

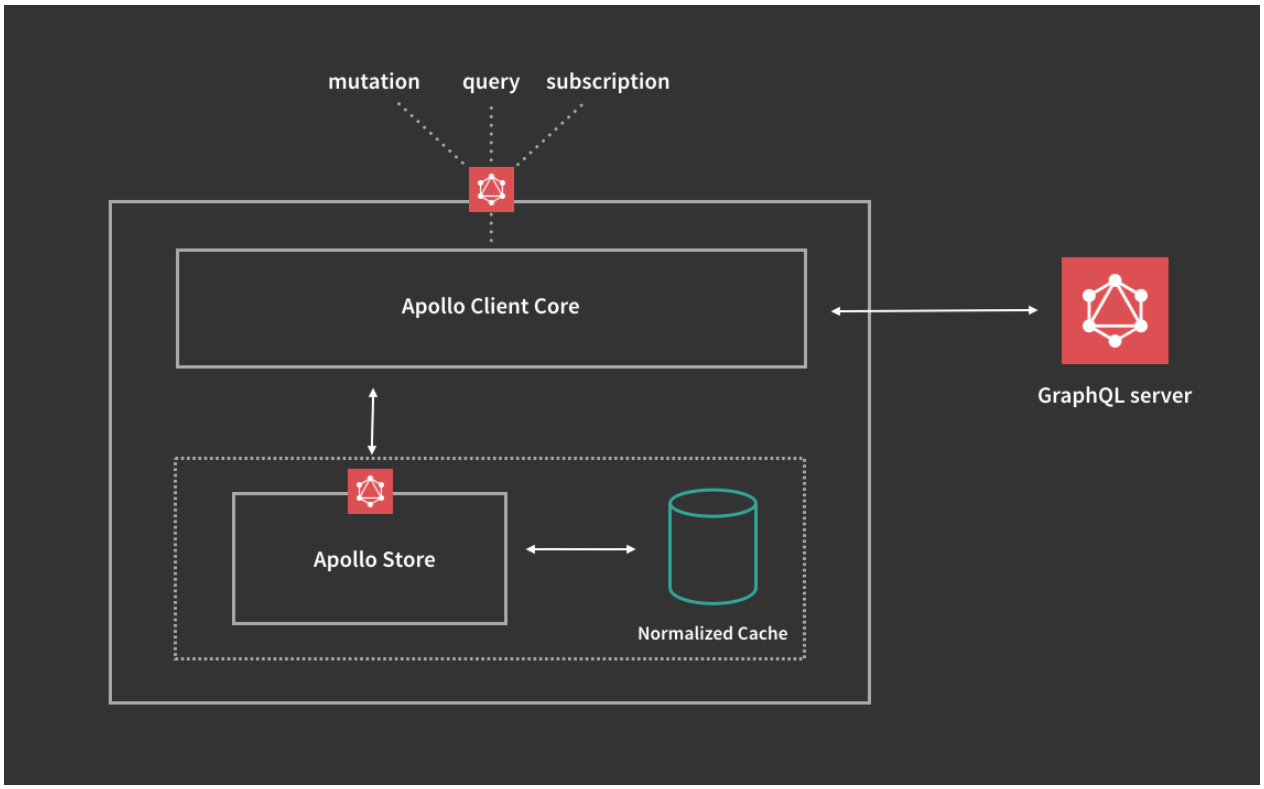
## 5.6 Aktualizace Apollo cache

### Specifikace a požadavky

Správné použití GraphQL spolu s Apollo Clientem je také práce s Apollo cache a její aktualizace. V případě, že z aplikace vyvoláme nějakou změnu či se přidá nějaký nový element, jsou dvě možnosti jak s touto skutečností pracovat.

První je znovu načtení GraphQL query (tzv. refetch), což obnáší opětovné volání query metody, jenž se dotazuje backendu a nedochází k úplnému využití cache. Tato možnost ovšem není nejlepší řešení, jelikož se provádí zbytečné dotazy právě na server.

Druhou možností je aktualizace cache přímo na staně klienta. Přímo při vyvolání nějaké akce (tzv. mutace), jenž vyvolá změnu v aplikaci je možné sáhnout do cache a přidat, odebrat či aktualizovat daný fragment dat či query.




Obrázek 5.9: Apollo cache [18]

### Analýza a návrh

V aplikaci probíhalo spoustu tzv. refetchování queries po různých změnách. Proto bylo nutné tyto případě revidovat a upravit na aktualizaci cache. Každá mutace poskytuje funkci **update** přes kterou je možnost si cache aktualizovat. V každém z těchto případů jsem si vždy zjistil, které query je třeba aktualizovat, následně jsem si vytvořil funkci, která bude tyto aktualizace obstarávat.

### Implementace

Jednalo se o poměrně jednoduché operace jako například práce s polem či odstraňování a aktualizování prvku v poli či objektu. Úkolem bylo docílit toho, aby z mutací zmizely všechny refetchQueries (viz. výpis č. 5.8), čímž docházelo k již zmíněnému znovu volání dotazů což způsobovalo zbytečný dotaz na server.

Na následujícím příkladu demonstruji ne úplně optimální a optimálnější způsob, jak aktualizovat onu cache. Jedná se o funkci pro odebrání uživatele z datové schránky. Z pohledu programátora je jednodušší použít způsob vyobrazený na výpisu č. 5.8, kdy se pouze znovu zavolá query a tím se aktualizují data - odebere se uživatel.

```
const onDeleteClick = useCallback(async () =>
    await deleteAuthorizedUser({
        variables: {
            dataMailboxUserId: props.id!,
        },
        refetchQueries: ['dataMailbox'],
    }), [deleteAuthorizedUser, props.id]);
```

Listing 5.8: Nevyužití úplného potenciálu Apollo cache

Toto ovšem není optimální, jelikož klademe vyšší nároky na server než je opravdu potřeba. Zároveň tímto způsobem není možná funkčnost i bez připojení k internetu, což řeší právě práce s cache na úrovni klienta. Ve výpise č. 5.9 je možno vidět využití funkce **deleteAuthorizedUserUpdater**. Tato funkce obstarává aktualizaci cache v offline režimu, tudíž není potřeba se znovu ptát serveru na data a pouze se upraví mezipaměť, aby odpovídala stavu jako po opětovném dotazu na server.

```
const onDeleteClick = useCallback(async () =>
    await deleteAuthorizedUser({
        variables: {
            dataMailboxUserId: props.id!,
        },
        update: deleteAuthorizedUserUpdater(props.id!),
    }), [deleteAuthorizedUser, props.id]);
```

Listing 5.9: Lepší využití Apollo cache

Funkce **deleteAuthorizedUserUpdater** pouze získala všechna potřebná data z mezipaměti pomocí pomocné funkce *readQuery*, kterou poskytuje Apollo client a následně vyfiltrovala uživatele, který byl odebrán z datové schránky. Tyto vyfiltrované data opětovně uložila do mezipaměti a nahradila tím stará data.

## 5.7 Tištění zpráv a jejich příloh

### Specifikace a požadavky

Uživatelé měli mít možnost vytištění přijaté zprávy co nejjednodušším způsobem - tj. kliknutí na tlačítko či stisknutí klávesové zkratky CTRL + P (MacOS COMMAND + P). Dále bylo potřeba zajistit tištění zprávy se seznamem příloh a také přílohy samostatné. Posledním požadavkem bylo tištění všech příloh najednou.

### Analýza a návrh

Funkcionalita měla být co nejpodobnější funkcionalitě využívající se v desktopové aplikaci MDS s ohledem na omezení webového prohlížeče. Funkcionalita vytištění zprávy se seznamem příloh, zprávy samotné bez seznamu a vytištění příloh samotných bylo již od pohledu jednoduché - pouhé vykreslení HTML kódu přijatého z backendu do iframe (element poskytnutý HTML5) a následné využití základní funkce prohlížeče pro tisk.

Problém nastal při analýze problému s vytištěním všech příloh. S kolegou jsme tento problém analyzovali a došli k závěru, že nebude možné vytvořit funkcionalitu vytištění všech příloh z důvodu, že webový prohlížeč tuto možnost neposkytuje a bylo by příliš komplikované toto omezení obejít a tuto funkcionalitu implementovat.

### Implementace

V první řadě jsem z grafického návrhu poskytnutého grafikem převzal styly a umístění tlačítek sloužících pro tištění jednotlivých prvků. Tlačítka jsem tedy umístil na předem určené místo.

V druhé řadě jsem po konzultaci s kolegou stanovil, že nejlepší možností jak implementovat tuto funkcionalitu je buď využití otevření nového okna s vykresleným obsahem či využití elementu iframe [19], který umožňuje vykreslení HTML dokumentu uvnitř nadřazeného HTML dokumentu. Rozhodl jsem se jít způsobem iframe, jelikož mi tento způsob přišel zajímavější a přijatenější z pozice UX pro uživatele.

Následně jsem vytvořil hook, jenž poskytoval funkcionalitu pro tištění zdroje iframe (atribut src) a také pro tištění HTML iframe (vložení HTML do obsahu iframe).

### Funkce pro vytištění HTML

Funkce **onPrintHTML** měla poměrně jednoduchou implementaci. Jednalo se o čistě TypeScriptovou funkci, která vytvořila iframe a přidala jej do těla HTML dokumentu. Následně vepsala potřebné HTML do obsahu iframe a nakonec využila funkci *resolveIFrame* vytvářející Promise [20], uvnitř kterého se nastavil *focus* na HTML dokument uvnitř iframe, který následně vytiskl pomocí funkce *print*.

```
const onPrintHTML = async (html) => {
        const iFrameToPrint = createIframe('iFrameHTML');
        const iFrameDocument = iFrameToPrint.contentDocument!;
        const iFrameWindow = iFrameToPrint.contentWindow!;

        writeHTMLIntoIFrame({ html, iFrameDocument });

        return resolveIFrame(iFrameWindow);
};
```

Listing 5.10: Funkce onPrintHTML

### Funkce pro vytištění SRC

Funkce **onPrintSRC** měla obdobnou implementaci jako funkce *onPrintHTML*, ovšem s rozdílem, že tato funkce potřebovala načíst soubor z backendu. Tuto skutečnost řešila funkce fetchSource, vracející URL tohoto souboru. Tato URL byla poté vepsána do iframe jako atribut src (zkrázenina pro source).

```
const onPrintSRC = async (src) => {
        const iFrameToPrint = createIframe('iFrameSRC');
        const iFrameDocument = iFrameToPrint.contentDocument!;
        const iFrameWindow = iFrameToPrint.contentWindow!;

        const url = await fetchSource(src!);

        if (url) {
            iFrameToPrint.src = url;
        } else {
            writeHTMLIntoIFrame({ NOT_SPECIFIED_HTML, iFrameDocument });
        }

        return resolveIFrame(iFrameWindow);
};
```

Listing 5.11: Funkce onPrintSRC

Po napsání těchto funkcí bylo možné napojit tuto funkcionalitu na již předem vytvořené tlačítka. Posledním úkolem bylo vytištění všech příloh najednou.

### Vytištění všech příloh

Pro vytištění všech příloh jsme s kolegou stráveným odpolednem našli řešení v asynchronním prováděním operace tisk pro jednotlivé přílohy. Každá příloha v tomto případě čekala na vytištění přílohy předešlé a proto bylo možné tuto funkcionalitu implementovat poměrně jednoduše což předčilo naše očekávání.

```
const printAllAttachments = async () => {
    if (attachments) {
        for (const attachment of attachments) {
            await onPrintSRC(
                createPreviewPath({
                    fileName: attachment.fileDescr,
                    token: attachment.downloadToken,
                })
            );
        }
};
```

Listing 5.12: Funkce pro vytištění všech příloh

# Kapitola 6

# Závěr

Z mého pohledu bylo veškeré očekávání výběrem praxe nadmíru uspokojeno. Absolvování odborné praxe pro mne bylo opravdu velkým přínosem, ze kterého jsem získal nesčetně praktických zkušeností, které nyní mohu uplatnit ve svém kariérním životě. Ovšem kromě čistě praktických zkušeností jsem spolupracoval s mnohými lidmi, díky němž jsem měl možnost se zdokonalovat ve svých schopnostech programátora ale také pro lidské a komunikační stránce.

Jeden z největších přínosů hodnotím agilní přístup k vývoji softwaru. V mnohýh velkých společnostech se již uplatňuje scrum master spolu s product ownerem, kteží zajišťují hladký průběh vývoje a odstiňují komunikaci obchodu s vývojáři. Fáze návrhu a analýzy byla z tohoto důvodu mnohem přínosnější a rychlejší, než při nevyužití agilního způsobu vývoje.

V této bakalářské práci byla popsána odborná praxe, jenž jsem absolvoval ve společnosti ATLAS consulting spol. s r.o. [3], kde jsem se podílel na vývoji a úpravách na produktech CODEXIS a MDS Online. Byly zde představeny technologie, se kterými jsem se dostal do styku, zejména React a JavaScript.

## 6.1 Uplatněné znalosti

Mezi uplatněné znalosti nabytými na akademické půdě bych určitě v první řadě zařadil ty teoretické, zejména algoritmizace, datových struktur a návrhových vzorů, které jsem měl možnost využití v průběhu mé odborné praxe. Určitě nesmím zapomenout na znalosti získané předměty zabývajícími se vývojem webových technologií jako například předmět Vývoj mobilních aplikací I či Vývoj internetových aplikací, kde jsem měl možnost se setkat s programovacím jazykem JavaScript [6] spolu s pokročilejšími technologiemi jako React a TypeScript [5]. Důležitá byla ovšem také práce s verzovacími nástroji, zejména Git, na který kladl důraz předmět Vývoj mobilních aplikací II.

Jako poslední věc je určitě nezbytná zmínka o předmětech Programovací jazyky I a II, které mi dopomohly k poznání programovacího paradigmatu a syntaxu a také předměr Algoritmy I a II, které byly součástí mého učení se o algoritmizaci.

## 6.2 Scházející znalosti

Ačkoliv jsem na akademické půdě nabyl majoritní většinu teoretických znalostí, scházelo mi právě spousta praktických znalostí.

Z počátku praxe bylo ode mne očekáváno dohnání scházejících znalostí co se týče agilního vývoje, což se mi povedlo po poměrně krátké době. Po určitém čase jsem byl schopen tyto, pro mě v určitou dobu, méně známé praktiky a způsoby vývoje ovládat a chápat na vyšší úrovni.

## 6.3 Shrnutí

Díky možnosti absolvovat tuto odbornou praxi ve firmě ATLAS consulting s. r.o jsem nabyl nezpočet zkušeností, které nyní mohu zúročit v profesním životě.

Mezi získané zkušenosti bych především zmínil knihovnu React a práci s programovacím jazykem TypeScript. Kromě specifických technologií, jsem měl možnost se rozvíjet v obecných principech programování a rozšířit svůj pohled na praktické zvyklosti a znalosti, dále schopnosti řešit přidělené problémy a schopnosti naučení neznámé technologie.

Velmi často jsem se potýkal s řešením více než pouze jednoho úkolu najednou, díky čemuž jsem se naučil organizace a disciplíny z pohledu dodání onoho řešení v určitém časovém horizontu.

# Literatura


1. *Codexis* [online] [cit. 2021-02-25]. Dostupné z: https://atlasconsulting.cz/software/codexis/.
2. *MDS - Online verze* [online] [cit. 2021-02-25]. Dostupné z: https://mdso.cz/#o-produktu.
3. *Atlas Consuling s r.o* [online] [cit. 2021-02-25]. Dostupné z: https://atlasconsulting.cz/.
4. *Atlas Consuling s r.o* [online] [cit. 2021-02-25]. Dostupné z: https://atlasconsulting.cz/o-nas/.
5. *TypeScript* [online] [cit. 2021-02-25]. Dostupné z: https://www.typescriptlang.org/.
6. *JavaScript* [online] [cit. 2021-02-25]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/About_JavaScript.
7. *JSX* [online] [cit. 2020-03-02]. Dostupné z: https://reactjs.org/docs/introducing-jsx.html.
8. *React komponenty* [online] [cit. 2021-03-01]. Dostupné z: https://reactjs.org/docs/components-and-props.html.
9. *React stav a životní cyklus* [online] [cit. 2021-03-01]. Dostupné z: https://reactjs.org/docs/state-and-lifecycle.html.
10. *React hooks* [online] [cit. 2021-03-01]. Dostupné z: https://reactjs.org/docs/hooks-intro.html.
11. *React hooks* [online] [cit. 2021-03-01]. Dostupné z: https://zdrojak.cz/clanky/react-hooks-ktere-potrebujete-znat/.
12. *GraphQL* [online] [cit. 2021-03-02]. Dostupné z: https://graphql.org/learn/.
13. *GraphQL vs Rest* [online] [cit. 2021-03-02]. Dostupné z: https://devopedia.org/graphql.
14. *Apollo* [online] [cit. 2021-03-02]. Dostupné z: https://www.apollographql.com/platform.
15. *Styled Components* [online] [cit. 2021-03-02]. Dostupné z: https://styled-components.com/docs.
16. *Codegen* [online] [cit. 2020-03-02]. Dostupné z: https://graphql-code-generator.com/docs/plugins/typescript-react-apollo.

17. *React Spring* [online] [cit. 2021-03-02]. Dostupné z: https://www.react-spring.io/.

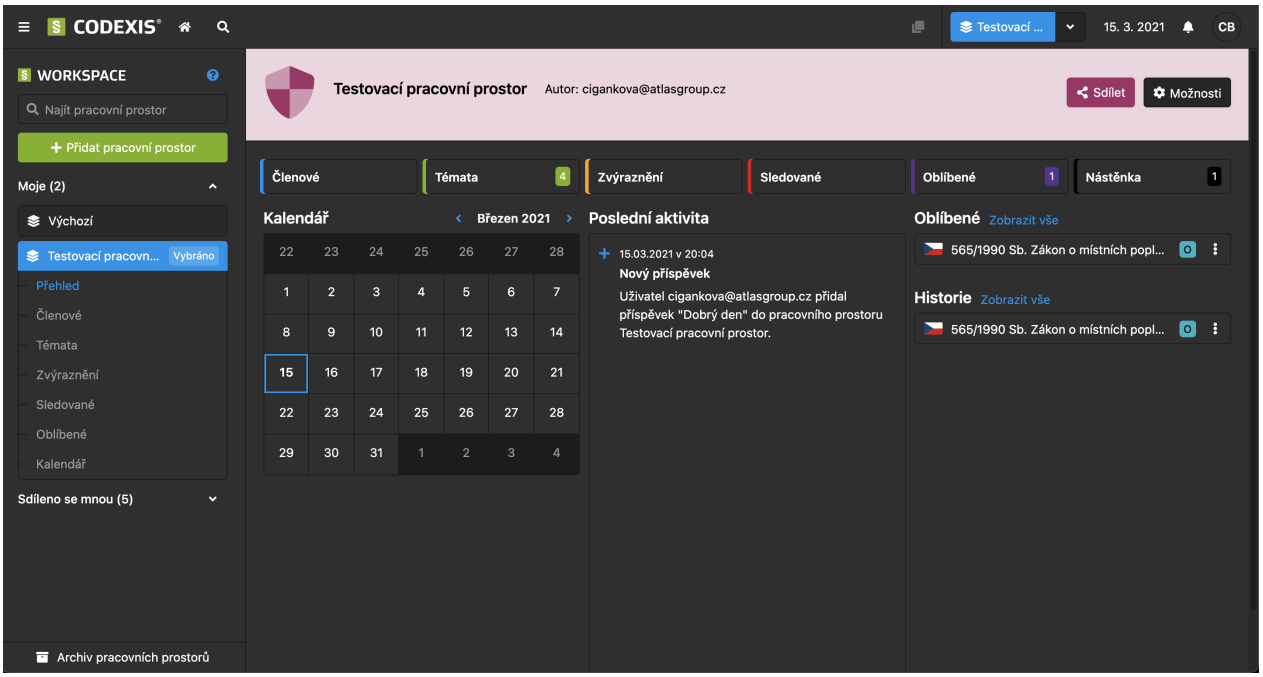
18. *CSS in JS* [online] [cit. 2020-03-02]. Dostupné z: https://miro.medium.com/max/3840/1*CbevjJN6IQBk7gbh38V-2g.png.

19. *iframe* [online] [cit. 2020-03-02]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTML/Element/iframe.

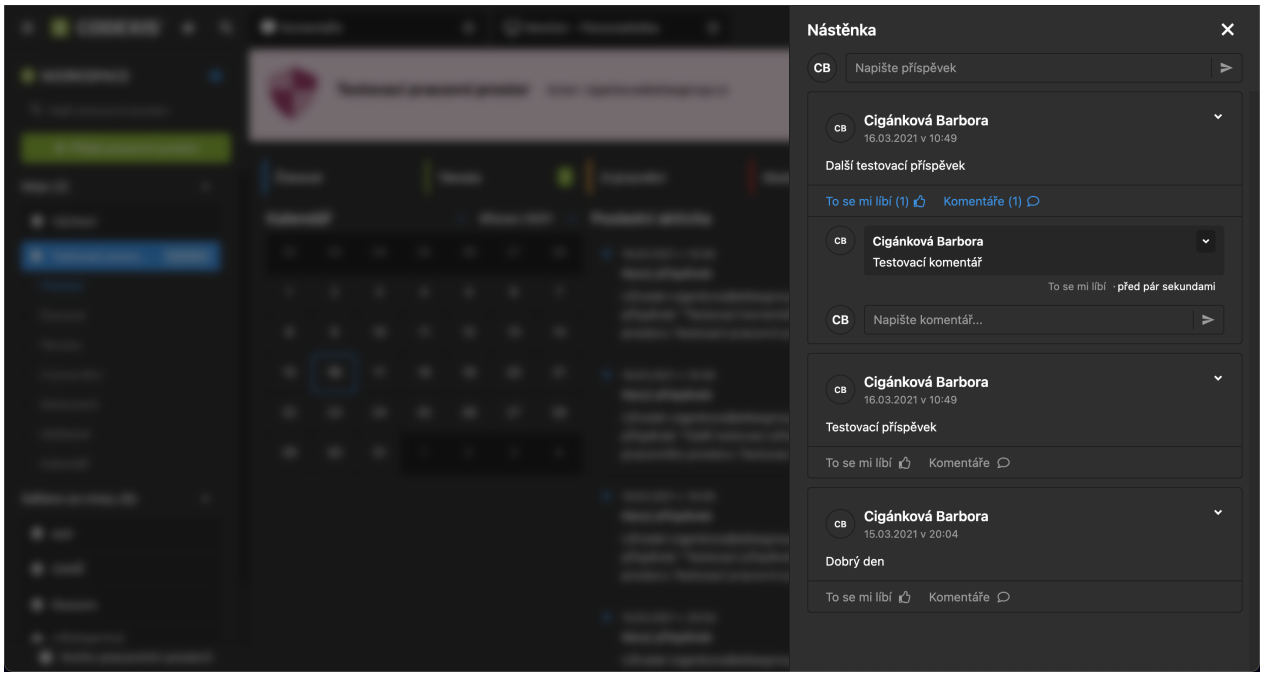
20. *promise* [online] [cit. 2020-03-02]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise.

# Příloha A

# Grafika pro Workspace



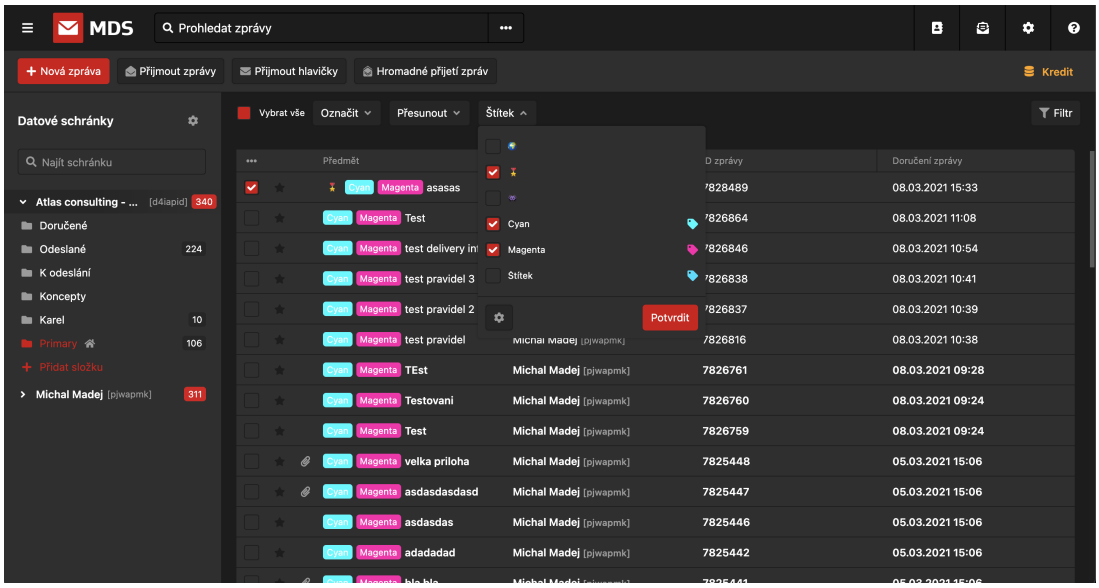

Obrázek A.1: Výsledek Workspace - Dashboard



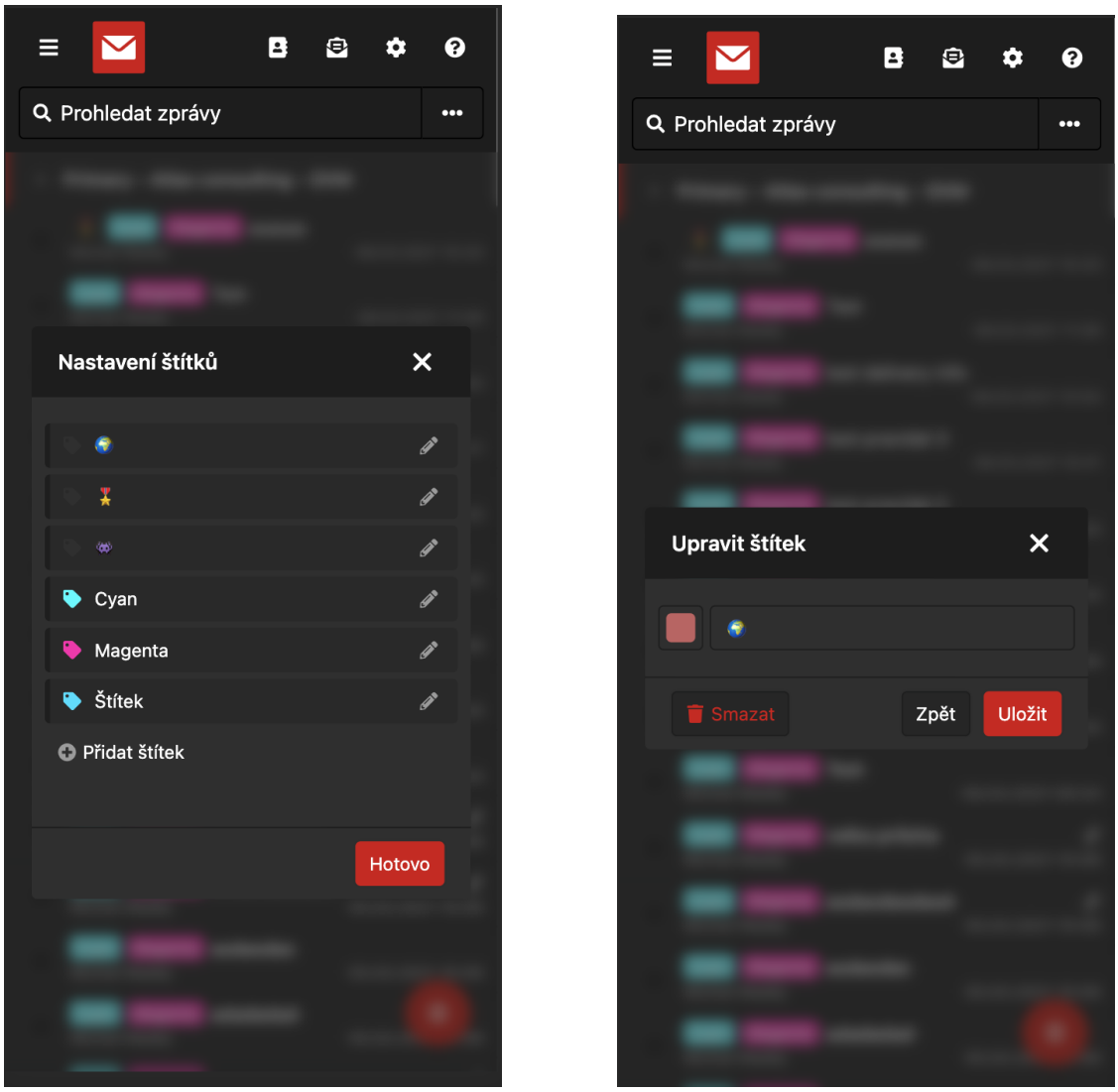

Obrázek A.2: Výsledek Workspace - Dashboard

# Příloha B

# Grafika pro štítky




Obrázek B.1: Výsledek štítků - Kontextové menu

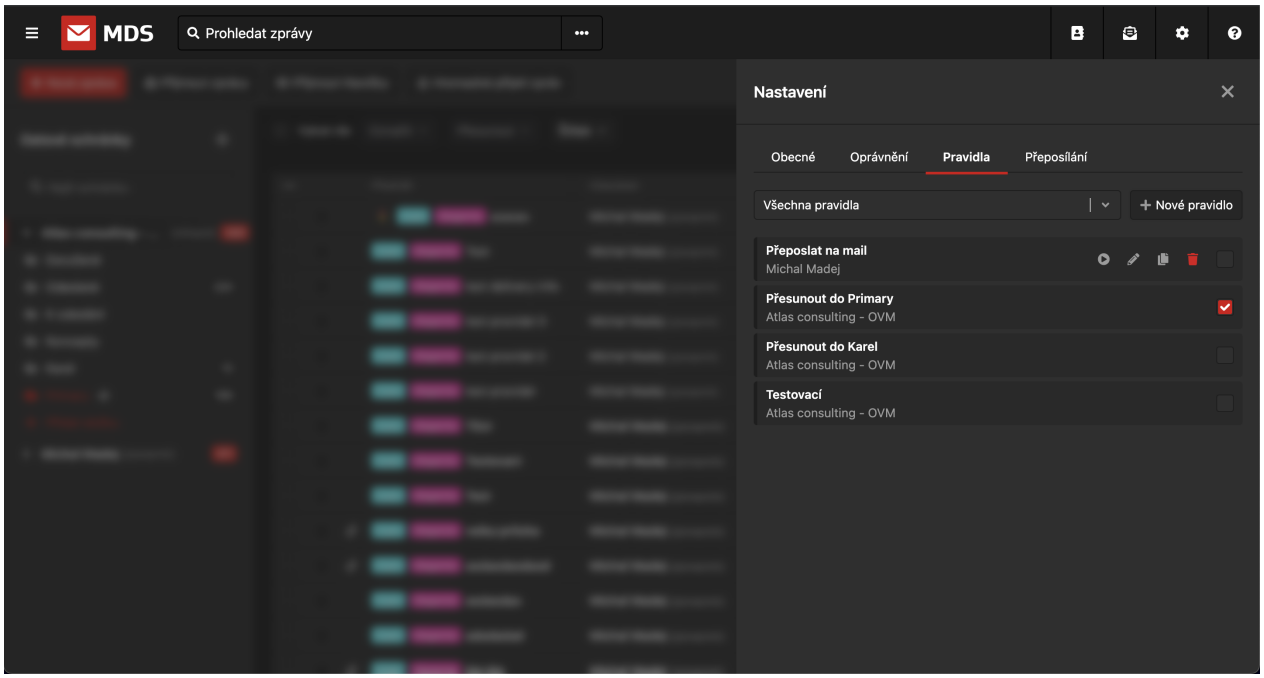



(a) výpis (b) editace

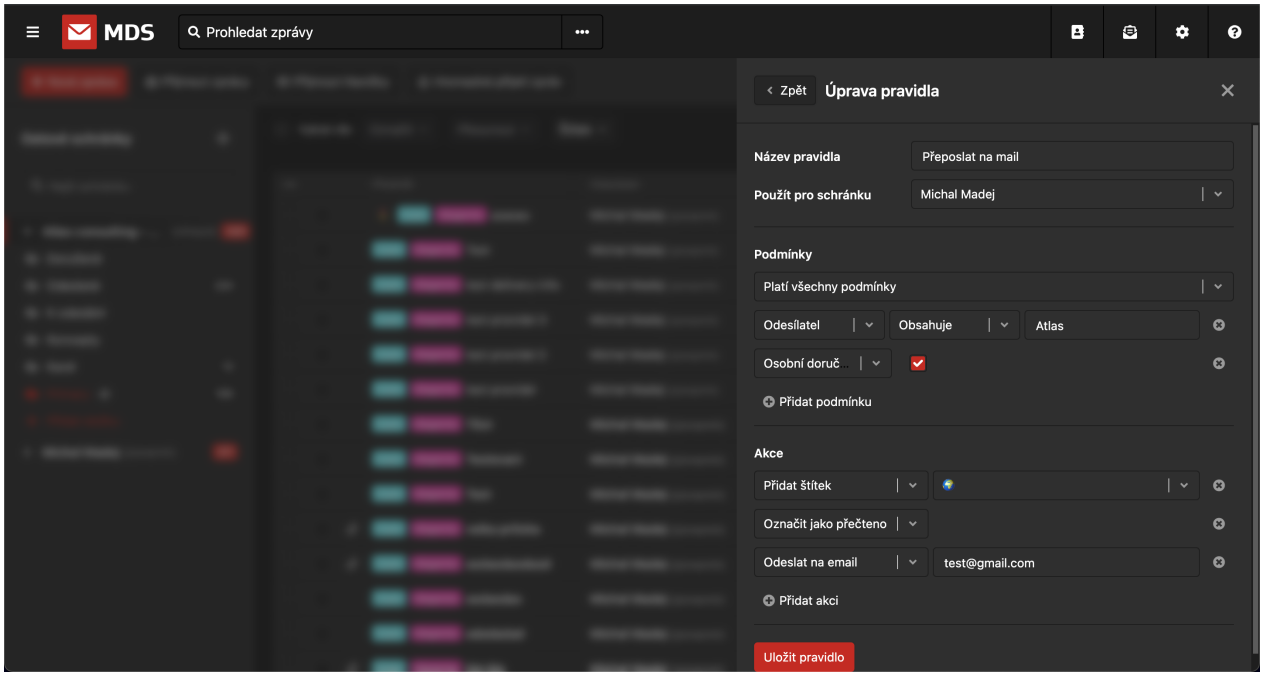
Obrázek B.2: Výsledek štítků - mobilní rozlišení

# Příloha C

# Grafika pro pravidla




Obrázek C.1: Výsledek pravidel - Výpis




Obrázek C.2: Výsledek pravidel - Nové  Editace